FEATURE ARTICLE

Hank Wallace

Using the Golay Error Detection and Correction Code

People often forget the work done long before the microprocessor when developing algorithms. The discoveries made by Marcel J.E. Golay in 1949 are well-suited for some of today's communications systems.

This is the scene: Smoke wafts slowly in the darkened room, as if with a mind of its own. Seedy characters sit at a dirty table under a naked bulb. At the focus, a Ouija board. Surrounding are piles of paper—data stacked upon data upon data, casting stark shadows. A sheet is snatched, examined, a scribble is placed on it, and then it's back to the pile. Occult bookies hard at work? No, error correction!

Well, that was my vision of it, until I was forced to bone-up for a recent communications project. I found out that error correction is not really like what you see in the movies: smoky mirrors, crystal balls, and evil mathematicians. If your vision of error correction is something to be avoided, I would like to put it in a better light and help you change that image.

Most engineers are familiar with basic methods to verify the integrity of data transmitted over noisy communications channels. Simple checksums and cyclic redundancy checks (CRCs) are the most popular. Presented here is a related coding technique by which errors cannot only be detected, but removed from received data without retransmissions. This code bears the name of its discoverer: Marcel J. E. Golay. The earliest reference to his work I found is a note he published in a 1949 technical journal [1]. Then, there was much activity in information coding techniques spurred by Claude Shannon's 1948 landmark work *The Mathematical Theory Of Communication*. The codes that Golay discovered can enhance the reliability of communication on a noisy data link.

Before we proceed, a note of caution. The theory of error detection and correction is deep (but not necessarily dark), as you can verify from any of the references cited. This article is intended only to give a brief overview of one of the many codes available. We will not dwell on messy mathematical details or terse comparisons of error correction techniques. If your system design calls for error correction methods, the C program fragments here may help you evaluate the Golay code. Just keep in mind that this is not a theoretically thorough treatment of this almost bottomless, fascinating subject.

We should get a couple of simple definitions out of the way to make what follows easier to digest.

Weight: The count of bits which are ones in a binary word. For example, the weight of the byte 01011011 is five since it contains five ones.

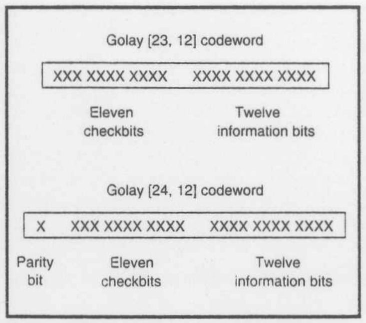


Figure 1—*The Golay codeword structure is derived from modulo-2 division, similar to the CRC.*

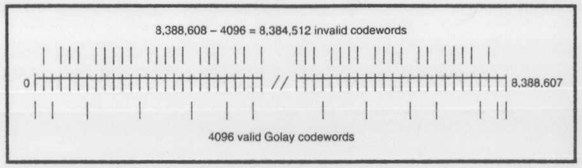


Figure 2—*Golay [23,12] codeword distribution on the number line spaced with respect to the Hamming distance.*

Hamming Distance: The number of bits which differ between two binary numbers. If A and B are binary numbers, the distance is weight(A XOR B). For example, the Hamming distance between bytes 4Ah and 68h is weight(4Ah XOR 6Fh) = weight(22h) = 2 bits.

Forward Error Correction: A technique which can correct errors at the receiver without relying on retransmissions of data.

NUTS AND BOLTS

Let's jump right in and examine the structure of the binary Golay code. A *codeword* is formed by taking 12 information bits and appending 11 check bits which are derived from a modulo-2 division, as with the CRC. See Figure 1. We will examine the modulo-2 division process later. The common notation for this structure is Golay [23,12], indicating that the code has 23 total bits, 12 information bits, and 23–12=11 check bits. Since each codeword is 23 bits long, there are 2^{23}, or 8,388,608, possible binary values. However, since each of the 12-bit information fields has only one corresponding set of 11 check bits, there are only 2^{12}, or 4096, valid Golay codewords. You can think of the codewords as being distributed along a number line. See Figure 2.

The Golay codewords are not really spread evenly as you count along the number line, say every few ticks. Rather, they are spaced with regard to the Hamming distance between them. It turns out that each Golay codeword has seven or more bits differing from every other. Mathematicians say that the code has a minimum distance, *d*, of seven. They have determined analytically that the Golay code can detect and correct a maximum of (d–1)/2=3 bit errors, in any pattern.

To augment the power of the Golay code, an overall parity bit is usually added, resulting in a clean 3-byte codeword called the *extended Golay code*, noted as Golay [24,12]. With this parity bit, all odd numbers of bit errors can be detected in each codeword, as well as all 4-bit errors. We'll get to the nuts and bolts of encoding and decoding codewords presently.

Before you trash all your old CRC-based designs, be advised that there is a tradeoff associated with the Golay code's error correction ability. If the code is used merely to detect errors, it can find a maximum six of them in any codeword. However, if correction is performed, only three bits are correctable. Thus, we trade identifiable errors for correctability. As the code is used in an application, situations may demand changing the correction/detection methods to suit. Keep this tradeoff in mind as you examine the performance of the Golay code and the requirements of your application.

Let's look at the error-trapping ability of the code. If error correction is not attempted, the following are the error-detection-only properties per 24-bit extended Golay codeword:

100% of 1- to 6-bit errors detected, any pattern
100% of odd bit errors detected, any pattern
99.988% of other errors detected

Using the error correction facilities of the code, these are the data reliability rates:

100% of 1- to 3-bit errors corrected, any pattern
100% of 4-bit errors detected, any pattern
100% of odd numbers of bit errors detected, any pattern
0.24% of other errors corrected (1/4096)

The parity bit of the extended Golay [24,12] code augments its error-corrective properties, allowing all combinations of 4-bit errors to be detected, but not corrected.

Error correction is obviously no panacea and does carry a penalty. Let me explain why. When a codeword is being corrected, the correction algorithm looks for the closest matching codeword. For example, say codeword E86555h is transmitted (data=555h, checkbits=E86h), but four bit errors occur, corrupting the codeword to E86476h. When we feed this codeword into the correction function, it returns 68E4E6h as the closest matching codeword, not E86555h. The receiver can use the parity bit to detect this error, but it cannot detect a higher even number of errors. This illustrates the fact that every correction algorithm has a bit-error-rate (BER) limit, beyond which it cannot compensate. Fortunately, you may enable the correction facilities of the given Golay C routines according to your needs.

Figure 3 illustrates what is happening during codeword correction. We see three valid Golay codewords: A, B, and C. The X-axis represents

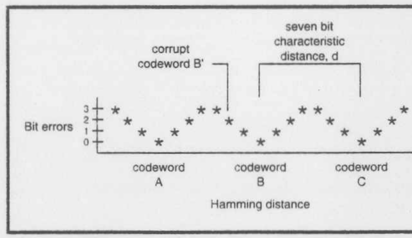


Figure 3—*Simplified illustration of Golay [23,12] error correction for three valid codewords.*

Hamming distance between these codewords. The Y-axis represents the number of bit errors incurred as you move away from a valid codeword. Picture the correction algorithm as taking the input codeword, say at point B', and sliding down the slope to the nearest correct codeword. If B' is a corrupted version of codeword B, we're in luck. If B' is a very corrupted version of A, the correction algorithm lies to us and returns B as the corrected codeword. You can see that the minimum distance, *d*, of the code controls the amount of corruption that can be tolerated. This is a simplified explanation, of course. In reality, this diagram is multidimensional and each Golay codeword has many error-laden companions to keep it company in the abstract boredom of *n*-space.

```
  Golay
polynomial          info bits     zero fill
|----------|     |-------------||-----------|
                 ________________________
101011100011    )10101010101000000000000
                 101011100011
                 ------------  <------------- XOR
                     100100100000
                     101011100011
                     ------------
                       111100001100
                       101011100011
                       ------------
                        101111011110
                        101011100011
                        ------------
                          100111101000
                          101011100011
                          ------------
                           01100001011 <---- checkbits
```

Figure 4—*Calculation of Golay checkbits using modulo-2 division.*

Now for some mathematical curiosities. Among error correction codes, there is a class known as *perfect codes*. Briefly, perfect codes are defined as those where each of the invalid codewords shown on the number line in Figure 2, when pumped through the correction process, will be transformed into a valid codeword. There are no *orphan uncorrectable information vectors*. Included as perfect codes are the *Hamming codes*, a one-bit correction scheme, and the *binary* and *ternary Golay* codes. The Golay codes are the only nontrivial multiple-error-correcting perfect codes that exist [3]. The perfect quality of the Golay code makes it an object of beauty in the eyes of mathematicians, especially when this property is expressed in terms of the optimum packing of spheres into a region of space [4]. This perfection also makes the Golay code useful in the hands of communications engineers.

The Golay codeword has some very interesting properties:

- Cyclic Invariance. If you take a 23-bit Golay codeword and cyclically shift it by any number of bits, the result is also a valid Golay codeword.
- Inversion. If you take a 23-bit Golay codeword and invert it, the result is also a valid Golay codeword.

Listing 1—*The Golay [23,12] codeword encoder returns a long integer with the format [checkbits(11), data(12)].*

```
#define POLY  0xAE3  /* or use the other polynomial, 0xC75 */

unsigned long golay(unsigned long cw)
{
  int i;
  unsigned long c;
  cw &= 0xfffl;
  c = cw;                          /* save original codeword */
  for (i=1; i<=12; i++) {          /* examine each data bit */
    if (cw & 1)                    /* test data bit */
      cw ^= POLY;                  /* XOR polynomial */
    cw >>= 1;                      /* shift intermediate result */
  }
  return((cw<<12) | c);            /* assemble codeword */
}
```

- Minimum Hamming Distance. The distance between any two Golay [23,12] codewords is always seven or more bits. The distance between any two Golay [24,12] codewords is always eight or more bits.
- Error Correction. The correction algorithm can detect and correct up to three bit errors per codeword.

WHAT'S THE USE?

The Golay code is obviously not able to encode a large amount of data in one codeword. Twelve bits is the maximum allowed. So what is its use? Well, one advantage to error correction is the elimination of communications retries which can bog down a noisy channel. In some cases, the overhead associated with a resend request is much greater than the length of a data packet. For example, to send a data packet on a half-duplex or simplex radio system, the microprocessor must turn on the transmitter and wait for it to come up to full power, typically 100 ms. The originating station must incur the same pretransmit delay when resending its data. Even if all goes well, there is 200 ms of wasted time involved in a resend request. At 1200 bits per second, 200 ms represents 240 bits of data, a reasonable message length in some systems. And if the radio channel is busy with other traffic, the delays can be longer. The

Listing 2—*The parity bit generator checks the overall parity of codeword cw. If parity is even, a 0 is returned; otherwise, a 1 is returned.*

```
int parity(unsigned long cw)

{
  unsigned char p;

  /* XOR the bytes of the codeword */
  p = *(unsigned char*)&cw;
  p ^= *((unsigned char*)&cw+1);
  p ^= *((unsigned char*)&cw+2);

  /* XOR the halves of the intermediate result */
  p = p ^ (p>>4);
  p = p ^ (p>>2);
  p = p ^ (p>>1);

  /* return the parity result */
  return(p & 1);
}
```

Golay code can reduce the number of retransmission events by allowing the receiving end to correct some errors in the received data, decreasing the probability that the channel will get overloaded.

In other situations, there may be no resend request possible (e.g., with a one-way infrared or ultrasonic data link). The Golay code allows such systems to increase the probability of one-way, error-free reception.

Some communications channels are more prone than others to burst errors, where many consecutive data bits are corrupted. The Golay code alone is not able to correct bursts of errors over three bits long in a single codeword. However, when augmented with a technique called *bit interleaving*, the Golay code's small, modular format allows the correction of large burst errors, as we will see. The unmodified Golay code really shines in applications prone to random bit errors though, and it tolerates a disgusting (3 bit errors)/(24 bits per codeword) = 12.5% bit error rate without data retransmissions.

Of course, a disadvantage of the extended Golay code is that you must transmit as many check bits as data bits. Maybe you are asking, why not just send the data twice, without check bits? The answer is that no error correction is possible in such a system. How would you know which of two different messages, if not both, were corrupt? The Golay code allows error correction in exchange for the data-doubling price.

IMPLEMENTATION—ENCODING

The Golay code is encoded just like the CRC, using modulo-2 division. However, there are only 12 information bits per codeword, so you must break your data down into 12-bit chunks and encode each as one codeword. The characteristic polynomials for the Golay code are:

1. $X^{11} + X^9 + X^7 + X^6 + X^5 + X + 1$, coefficients AE3h.
2. $X^{11} + X^{10} + X^6 + X^5 + X^4 + X^2 + 1$, coefficients C75h.

Yes, there are two. You only need to use one of them because they both have the same properties as mentioned above, though they generate differing checkbits. Pull out your favorite buffalo nickel and choose a polynomial! We will use the coefficients of the X terms of the first polynomial, AE3h. (Note that AE3h is the bit-reversed version of C75h. There is a greater power at work here.)

An example worked out by hand illustrates the modulo-2 division process. Remember that in modulo-2 division, we XOR instead of subtract. The data we wish to encode is 555h. To generate the 11 check bits, append 11 zeros onto the bit-reversed data (least-significant bit first) and perform the division in Figure 4.

We only care about the bit-reversed remainder from the division, 11010000110 = 686h, so do not even write down the quotient. Putting the codeword together for transmission, we get 686555h. A parity bit could be added to the codeword to make an extended Golay code. Of course, you can mish-mash the bits around in any order for transmission, just as long as they are reassembled correctly before doing the decoding.

The encoding algorithm is shown in Listing 1. Long integers are used to conveniently store one codeword. The routine `parity()` adds the parity bit to complete the extended codeword. It is shown in Listing 2. You can see reference 3 for matrix methods which encode a Golay [24,12] codeword directly without an explicit parity routine.

IMPLEMENTATION—DETECTING ERRORS

Detection of errors in a codeword is easy. First, we use the overall parity bit to check for odd numbers of errors, possibly failing the codeword on that basis. If parity is correct, we compute the 23-bit syndrome of the codeword and check for zero. The syndrome is the remainder left after the codeword

```
  Golay
polynomial        info bits     check bits
|----------|     |----------||----------|
                 _____________________________
101011100011    )10101010101001100001011
                 101011100011
                 ------------
                      100100101100
                      10101?100011
                      ------------
                        111100111100
                        101011100011
                        ------------
                         101110111111
                         101011100011
                         ------------
                           101011100011
                           101011100011
                           ------------
                            00000000000 <---- syndrome = 0
```

Figure 5—*The syndrome is the remainder left after the codeword is divided by the generating polynomial, modulo-2. If the Golay codeword is valid, the syndrome is zero.*

Listing 3—*A Golay [23.12] codeword syndrome generator. A table of intermediate results could also be used to speed the process.*

```
unsigned long syndrome(unsigned long cw)

{
  int i:
  cw &= 0x7fffffl:
  for (i=1: i<=12: i++) { /* examine each data bit */
    if (cw & 1)           /* test data bit */
      cw ^= POLY:         /* XOR polynomial */
    cw >>= 1:             /* shift intermediate result */
  }
  return(cw<<12):         /* value pairs with upper bits of cw */
}
```

Listing 4—*Golay [23,12] codeword correction routines.*

```
int weight(unsigned long cw)

/* Calculate the weight of 23-bit codeword cw. */
{
  int bits.k:

  /* nibble weight table */
  const char wgt[16] = {0.1.1.2.1.2.2.3.1.2.2.3.2.3.3.4}:

  bits = 0; /* bit counter */
  k = 0:

  /* do all bits. six nibbles max */
  while ((k<6) && (cw)) {
    bits = bits+wgt[cw & 0xf]:
    cw >>= 4:
    k++:
  }
  return(bits):
}


unsigned long rotate_left(unsigned long cw. int n)

/* Rotate 23-bit codeword cw left by n bits. */
{
  int i:

  if (n != 0) {
    for (i=1: i<=n: i++) {
      if ((cw & 0x4000001) != 0)
        cw = (cw << 1) | 1:
      else
        cw <<= 1:
    }
  }
  return(cw & 0x7fffffl):
}


unsigned long rotate_right(unsigned long cw. int n)

/* Rotate 23-bit codeword cw right by n bits. */
{
  int i:

  if (n != 0)
    {
      for (i=1: i<=n: i++)
        {
          if ((cw & 1) != 0)
            cw=(cw >> 1) | 0x4000001:
          else
            cw>>=1:
        }
    }
  return(cw & 0x7fffffl):
}


unsigned long correct(unsigned long cw. int *errs)

/* Correct Golay [23.12] codeword cw. returning the corrected
codeword. This function will produce the corrected codeword for
three or fewer errors. It will produce some other valid Golay
codeword for four or more errors. possibly not the intended one.
*errs is set to the number of bit errors corrected. */
```

(continued)

is divided by the generating polynomial, modulo-2. If the codeword is a valid Golay codeword, the syndrome is zero, otherwise it will be nonzero. Figure 5 shows a sample calculation using the previously constructed codeword, 686555h, bit reversed for the division process.

The routine in Listing 3 does the same sort of calculation. You could use a table of intermediate results to speed the process. These routines are all you need to implement a Golay error detection scheme. As above, you can detect up to six bit errors per codeword and, with the parity bit, all odd numbers of errors.

IMPLEMENTATION—CORRECTING ERRORS

Cue the smoke and mirrors; error correction is not so trivial. It relies on the cyclic invariant properties of Golay codewords in a scheme called *systematic search decoding* [2]. There are other methods of Golay error correction listed in the references, though I have found this one easy to implement in software. This is a "nearest neighbor decoder," because it determines which Golay codeword is closest in terms of Hamming distance.

Here is a sketch of the systematic search algorithm.

1. Compute the syndrome of the codeword. If zero, no errors exist so go to step 5. If a trial bit has been toggled, go to step 2a, else go to 2.

2. If the syndrome has a weight of three or less, the syndrome matches the error pattern bit-for-bit and can be used to XOR the errors out of the codeword; if so, remove the errors and go to step 5, otherwise proceed to step 3.

2a. If the syndrome has a weight of one or two, the syndrome matches the error pattern bit-for-bit and can be used to XOR the errors out of the codeword; if so, remove the errors and go to step 5, else proceed.

3. Toggle a trial bit in the codeword in an effort to eliminate one bit error. Restore any previously toggled trial bit. If all 23 bits have been toggled and tried once, go to step 4; else go to step 2a.

Listing 4—*continued*

```
{
  unsigned char w:  /* current syndrome limit weight. 2 or 3 */
  unsigned long mask:  /* mask for bit flipping */
  int i,j:              /* index */
  unsigned long s,   /* calculated syndrome */
   cwsaver:             /* saves initial value of cw */

  cwsaver = cw:       /* save */
  *errs = 0:
  w = 3:                /* initial syndrome weight threshold */
  j = -1:               /* -1 = no trial bit flipping on first pass */
  mask = 1:

  while (j < 23) {           /* flip each trial bit */
    if (j != -1) {           /* toggle a trial bit */
      if (j > 0) {           /* restore last trial bit */
        cw = cwsaver ^ mask:
        mask += mask:        /* point to next bit */
      }
      cw = cwsaver ^ mask:  /* flip next trial bit */
      w = 2:          /* lower the threshold while bit diddling */
    }

    s = syndrome(cw):                /* look for errors */

    if (s) {                         /* errors exist */
      for (i=0: i<23: i++) {
        /* check syndrome of each cyclic shift */
        if ((*errs=weight(s)) <= w) {
          /* syndrome matches error pattern */
          cw = cw ^ s:                /* remove errors */
          cw = rotate_right(cw,i):  /* unrotate data */
          return(s=cw):
        }
        else {
          cw = rotate_left(cw,1):   /*  next pattern */
          s = syndrome(cw):          /* calc new syndrome */
        }
      }
      j++:                 /* toggle next trial bit */
    }
    else
      return(cw):          /* return corrected codeword */
  }
  return(cwsaver):         /* return original if no corrections */
} /* correct */
```

4. Rotate the codeword cyclically left by one bit. Go to step 1.

5. Rotate the codeword right to its original position, if needed.

The idea is to fiddle with the codeword until the syndrome has a weight of three or less, in which case we can XOR the syndrome with the codeword to negate the errors. However, if a trial bit has been toggled, we might have introduced an error (making a total of four), so the threshold for XORing the errors away in step 2a must be reduced by one for safety, to two or less.

We are relying on the perfect nature of the Golay code for a terminating condition of the search algorithm. Normally, we would test in step 4 to see if all 23 codeword shifts had already been tried, but since the Golay code is perfect, we know that each 23-bit value maps to exactly one correct Golay codeword and the algorithm will terminate. However, since programmers are not perfect, I test to guard against infinite loops.

Listing 5—*Golay [24,12] codeword decoder. This function decodes codeword *cw in one of two modes. If correct_mode is nonzero, error correction is attempted, with *errs set to the number of bits corrected, and returning 0 if no errors exist or 1 if parity errors exist. If correct_mode is zero, error detection is performed on *cw, returning 0 if no errors exist, 1 if an overall parity error exists, and 2 if a codeword error exists.*

```
int decode(int correct_mode, int *errs, unsigned long *cw)

{
  unsigned long parity_bit:

  if (correct_mode) {                 /* correct errors */
     parity_bit = *cw & 0x8000001: /* save parity bit */
     *cw &= ~0x8000001: /* remove parity bit for correction */

     *cw=correct(*cw, errs):     /* correct up to three bits */
     *cw|=parity_bit:            /* restore parity bit */

     /* check for 4 bit errors */
     if (parity(*cw))            /* odd parity is an error */
       return(1):
     return(0): /* no errors */
  }

  else { /* detect errors only */
    *errs=0:
    if (parity(*cw)) { /* odd parity is an error */
      *errs=1:
      return(1):
    }
    if (syndrome(*cw)) {
      *errs=1:
      return(2):
    }
    else
      return(0): /* no errors */
  }

} /* decode */
```

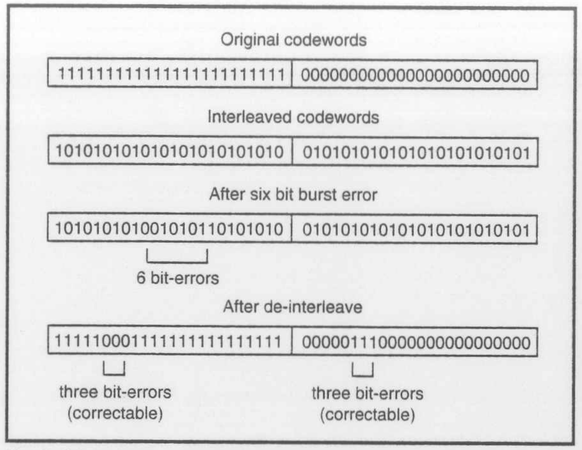


Figure 6—*Bit interleaving can be used in the presence of burst errors to extend the effectiveness of the Golay method.*

The rotations and trial-bit bookkeeping make for a nontrivial-looking routine for Golay error correction, `correct()`, shown in Listing 4 with its support routines. These routines are written for simplicity. You can tighten them up for optimum performance on your favorite scream machine.

The `correct()` function returns the nearest matching Golay codeword. Note that this may not be the codeword you expect if there are more than three bit errors because the received, corrupted codeword may be closer, in terms of Hamming distance, to a different Golay codeword.

Listing 5 shows `decode()`, a function that handles either detecting or correcting errors in Golay [24,12] codewords, returning the corrected codeword and error detection status upon exit.

The parity bit in the extended Golay code is used as a final check on the integrity of the corrected codeword. If the received codeword has the right parity, the codeword is accepted, otherwise it is rejected. The overall parity bit is used to trap the occurrence of odd numbers of errors and four-bit-error-laden codewords. That the overall parity bit traps four-bit errors is explained by the fact that the correction algorithm introduces three additional bit changes to get to the nearest codeword, which is seven bits distant from the original, noncorrupt codeword. So, the four-bit errors, plus three additional changed bits, make seven bit changes total.

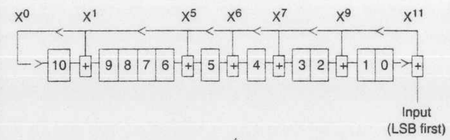
Note that toggling any bit(s) in a codeword an odd number of times will change the codeword's parity, allowing the overall parity check to trap four-bit errors, but only after correction has been attempted. If one of the four errors is in the parity bit, the 23-bit codeword will be corrected properly, but must be trashed because the total parity is wrong and the receiver will not know exactly which bits were corrupted. For the interested few, this will occur in $[C(24,4)-C(23,4)]/C(24,4) = (10626-8855)/10626 = 1/6$ of cases of four-bit errors, where $C(n,r)$ is the number of combinations of n things taken r at a time.

BACK TO THE REALITY

In practice, I have used the Golay code on radio channels and telephone lines and have found it to be robust. Practically, one cannot rely on the error correction facilities alone if there is a two-way communication link. Rather, I use the Golay code to reduce the incidence of message resends. Messages are always checked for framing errors and other problems which are specific to the information transmitted and the message context.

A frailty of the Golay code is its inability to detect large bursts of bit errors. However, a technique called *bit interleaving* can remedy the situation. Assume we have a block of 48 bits to transmit—two codewords. An error of four consecutive bits will corrupt a Golay data packet and elicit a retransmission. However, if we transmit the 48 bits not in two sequential Golay codewords, but in a scrambled order, burst errors will be distributed over both codewords when the bits are unscrambled. See Figure 6.

The two resulting codewords in Figure 6 are correctable, whereas a six-bit burst error in the original data would not have been. In general, the more data you interleave, the longer the burst of errors the message can withstand. The correctable burst size, in bits, is three times the number of codewords interleaved. The penalty is the reception delay incurred between the first and last bit of any one codeword. Using this method, it is theoretically possible to extend the 3/24=12.5% burst error correction capability of the Golay [23,12] code to arbitrarily long blocks of data. Imagine sending a 100-codeword, 2400-bit message and correcting a burst error of 300 consecutive bits! Unfortunately, interleaving data in this fashion is not fun or memory efficient, but that's life.



Shift register Bit 10 — Shift register Bit 0 — input data, LSB first

		register value	input data
initial register value —>		00000000000	
value after shift	1	11000111010	1
	2	01100011101	0
	3	00110001110	1
	4	00011000111	0
	5	00001100011	1
	6	11000001011	0
	7	01100000101	1
	8	11110111000	0
	9	10111100110	1
	10	01011110011	0
	11	00101111001	1
	12	11010000110	0

Figure 7—*Golay checkbit generator and example for polynomial $x^{11} + x^9 + x^7 + x^6 + x^5 + x + 1$. Modulo-2 division example is shown in Figure 5.*

You may have noticed that the codewords used in the above examples are 000000h and FFFFFFh. These are valid Golay codewords. This alerts us to a possible communications failure mode: the stuck bit. If the hardware were to fail in midmessage, it could emit a default 1 or 0, depending on its configuration and the data format. This could be disastrous if undetected because the dead circuit would be interpreted as sending consecutive, valid Golay codewords. But happily, we can modify the Golay checkbits slightly, as is often done when using CRCs, by inverting one or more of them before transmission, and reinverting the same bits before decoding at the receiver. This way, 000000h and FFFFFFh are not valid Golay codewords, and the chances of a stuck data line mimicking good data are greatly reduced. In fact, it is always good practice to structure your communications protocol so that a continuous 0 or 1 block of data cannot be misconstrued as valid information.

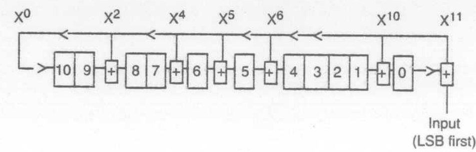
By the way, do not ignore the information that spins off of the correction process: The number of errors corrected in each codeword. This is valuable information about the BER on the communications channel, useful for judging its quality. Say you have a data system running over the telephone network in a remote area and these lines are not the best quality, degrading some with the weather. This is not a contrived example. Some Mom and Pop telcos run with twine and tin cans. It would be nice to optimize the data rate based on the current condition of the lines. Counting corrected bit errors is one way to do this. As the error rate (corrected bits / total bits) passes a critical value, the data rate can be cut, with the cooperation of transmitter and receiver, of course. When the sun comes out, the data rate can be increased as the BER goes down to acceptable levels. (Note that this technique has a BER measurement ceiling of 12.5% because every corrupt codeword will have at most three correctable errors.)

I am sure some readers are hardware types, wondering if the Golay code can be easily implemented in hardware. One manufacturer I know of makes a Golay encoder/decoder in IC form: Space Research Technology (Houston, Tex., [713] 782-2244). This device is amenable to serial data streams of Golay encoded data. They also publish some application notes mathematically detailing the advantages of bit interleaving and the Golay code in general.

The references for this article describe various circuits suitable to the task of decoding—I mean suitable as a function of your personal Boolean logic overload level. The circuits are not trivial by any means, though the relatively small number of 4096 Golay codewords suggests that ROM-based decoders may be economical. However, the shift register implementations of the two Golay checkbit generators are simple, and are shown in Figures 7 and 8. As before, either circuit produces valid Golay checkbits. Also shown is a sample calculation for each. The final register value in Figure 7 is 11010000110, which we reverse and compare with the result of Figure 4, finding agreement. This circuit is equivalent to that calculation with input data of 555h. The final register value of Figure 8 is 10011110100,

Hand verification of shift register calculation

```
110001110101 )10101010101000000000000
              110001110101
              ------------
               110110111110
               110001110101
               ------------
                  111001011000
                  110001110101
                  ------------
                    100010110100
                    110001110101
                    ------------
                     100110000010
                     110001110101
                     ------------
                      101111101110
                      110001110101
                      ------------
                       111100110110
                       110001110101
                       ------------
                         110100001100
                         110001110101
                         ------------
                          00101111001
```



Shift register Bit 10 — Shift register Bit 0 — input data, LSB first

		register value	input data
initial register value —>		00000000000	
value after shift	1	10101110001	1
	2	11111001001	0
	3	01111100100	1
	4	00111110010	0
	5	10110001000	1
	6	01011000100	0
	7	10000010011	1
	8	11101111000	0
	9	11011001101	1
	10	11000010111	0
	11	01100001011	1
	12	10011110100	0

Figure 8—*Golay checkbit generator and example for polynomial $x^{11} + x^{10} + x^6 + x^5 + x^4 + x^2 + 1$.*

which agrees with the accompanying modulo-2 division remainder, when reversed. Also notice that, like the two polynomials, the shift register circuits are mirror images of each other, suggesting that some more gating could make one bidirectional shift register encode either polynomial.

CONCLUSION

Of course, there are other error-correcting codes, but it is generally agreed in the literature that they are more difficult to correct than the Golay code, though the Hamming codes are easier. In general, the more capable the code, the more processor power it takes to decode. The Golay code could be used in even the smallest microcontrollers, such as the PIC series and 68HC05.

As a parting imperative, do your homework before choosing this or any other error detection/correction code. There are some important ideas that I did not discuss here, such as coding gain. This is an expression of the improvement in performance of the coded channel over the uncoded channel. Each code has a different coding gain for various channel noise conditions. You can also examine the references for matrix methods of encoding and decoding the Golay and other codes.

Finally, let me recommend the routine `golay_test()` in Listing 6. It does tests to verify the operation of your encoding and decoding routines by letting you induce a selectable number of errors in a test codeword on which correction is attempted. Always test your versions of these routines thoroughly before deploying; it is easy to slip up on a loop index and get a data-dependent error detector or corrector, which is certain to ruin a couple of nights' sleep.

Listing 6—*Golay [23,12] codeword test routine. This function tests the Golay routines for detection and correction of various patterns of error_limit bit errors. The error_mask cycles over all possible values, and error_limit selects the maximum number of induced errors.*

```
void golay_test(void)

{
  unsigned long
    error_mask,       /* bitwise mask for inducing errors */
    trashed_codeword,/* the codeword for trial correction */
    virgin_codeword; /* the original codeword without errors */

  unsigned char
    pass=1,           /* assume test passes */
    error_limit=3;    /* select number of induced bit errors here */

  int
    error_count;      /* receives number of errors corrected */

  virgin_codeword=golay(0x555); /* make a test codeword */

  if (parity(virgin_codeword))
    virgin_codeword^=0x800000l;

  for (error_mask=0; error_mask<0x800000l; error_mask++) {
    /* filter the mask for the selected number of bit errors */
    if (weight(error_mask) <= error_limit) {
      /* you can make this faster! */
      trashed_codeword=virgin_codeword ^ error_mask;
      /* induce bit errors */
      decode(1, &error_count, &trashed_codeword);
      /* try to correct bit errors */
      if (trashed_codeword ^ virgin_codeword) {
        printf("Unable to correct %d error mask induced = 0x%lX\n",
                weight(error_mask), error_mask);
        pass = 0;
      }

      if (kbhit()) {   /* look for user input */
        if (getch() == 27)
          return;        /* escape exits */
        /* other key prints status */
        printf("Current test count = %ld of %ld\n", error_mask,
                0x800000l);
      }
    }
  }

  printf("Golay test %s!\n", pass ? "PASSED" : "FAILED");
}
```

Hank Wallace is the owner of Atlantic Quality Design Inc., an embedded systems hardware and software design firm located in Rural Hall, N.C. He may be reached at (910) 377-2843 or hwallace@cybernetics.com.

REFERENCES

1. Golay, Marcel J.E., "Notes on Digital Coding," *Proceedings of the IRE*, Vol. 37, June 1949, pp. 657.
2. Lin, Shu, *An Introduction to Error Correcting Codes*. Englewood Cliffs, N.J., Prentice-Hall, 1970, pp. 101–106.
3. MacWilliams, Sloane, *The Theory of Error Correcting Codes*. Amsterdam, North Holland Publishing Co., 1977, pp. 64–65, 482, 634–635.
4. Pless, Vera, *Introduction to the Theory of Error Correcting Codes*. John Wiley & Sons, 1982, pp. 1–13.
5. Shannon, C.E., "A Mathematical Theory of Communication," *Bell System Technical Journal*, Vol. 27, July 1948, pp. 379–423; October 1948, pp. 623–656.

IRS

404 Very Useful
405 Moderately Useful
406 Not Useful